co

Assignment 1

Distributed System Design

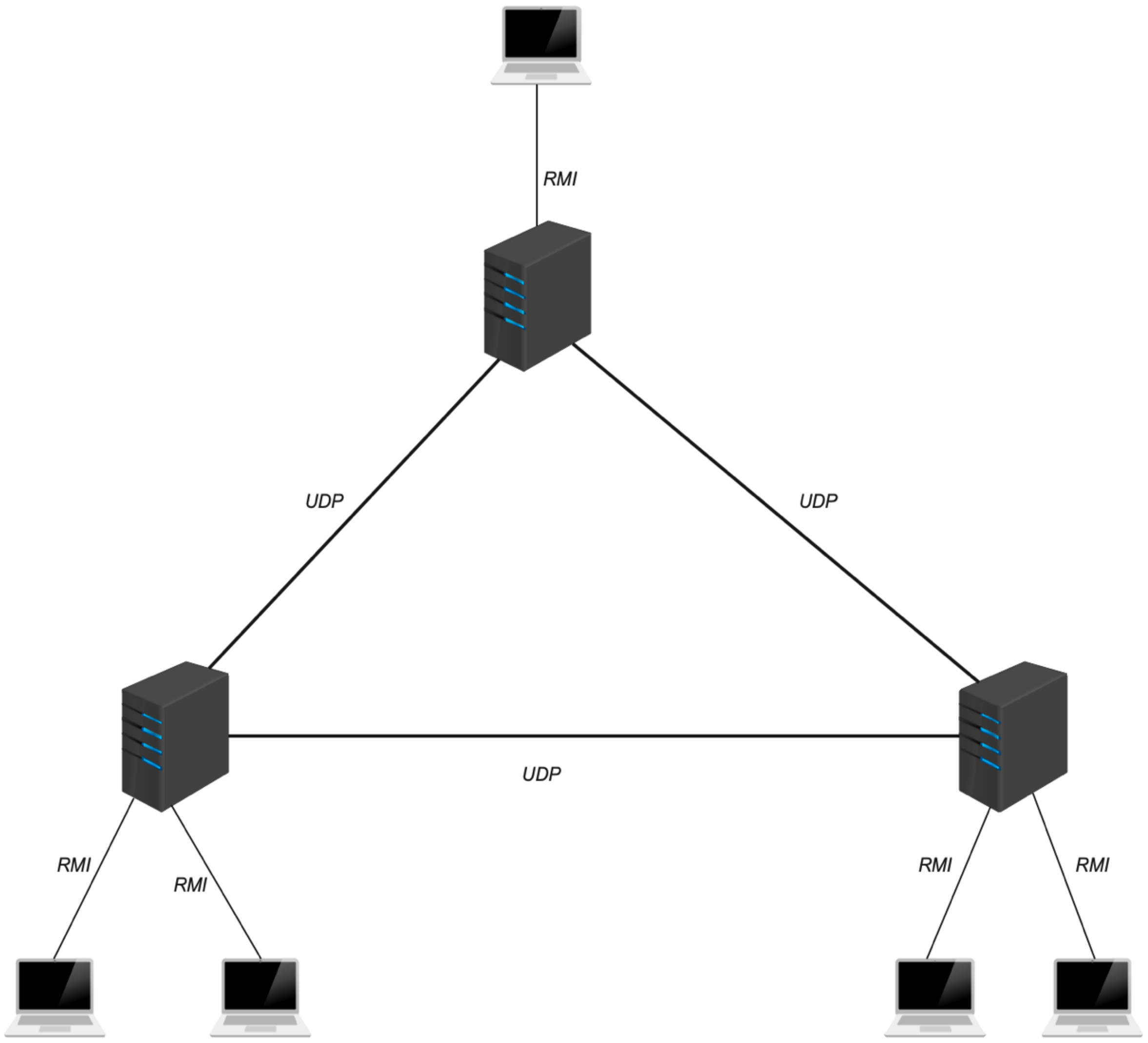
Concordia University

Peyman Shobeiri

ID: 40327586

Winter 2025

For this assignment, we were asked to create a distributed system for share management. This market includes three different cities and some clients who could buy shares, or perform different operations in the system. To explain the problem, we have three servers each of them is located in one of the cities. Each of these servers has an admin who manages its server and can call some methods which will be explained later. Each server also has multiple clients that interact with it. The servers are connected to each other using UDP protocol and the clients of each server are connected to the server using the RMI protocol. As mentioned in the question, there are some conditions such as each server admin could only manage its server. Additionally, if a client wants to purchase some shares from another city/server it should be able to perform this operation. An overview and a general idea of the problem is shown in the figure below:



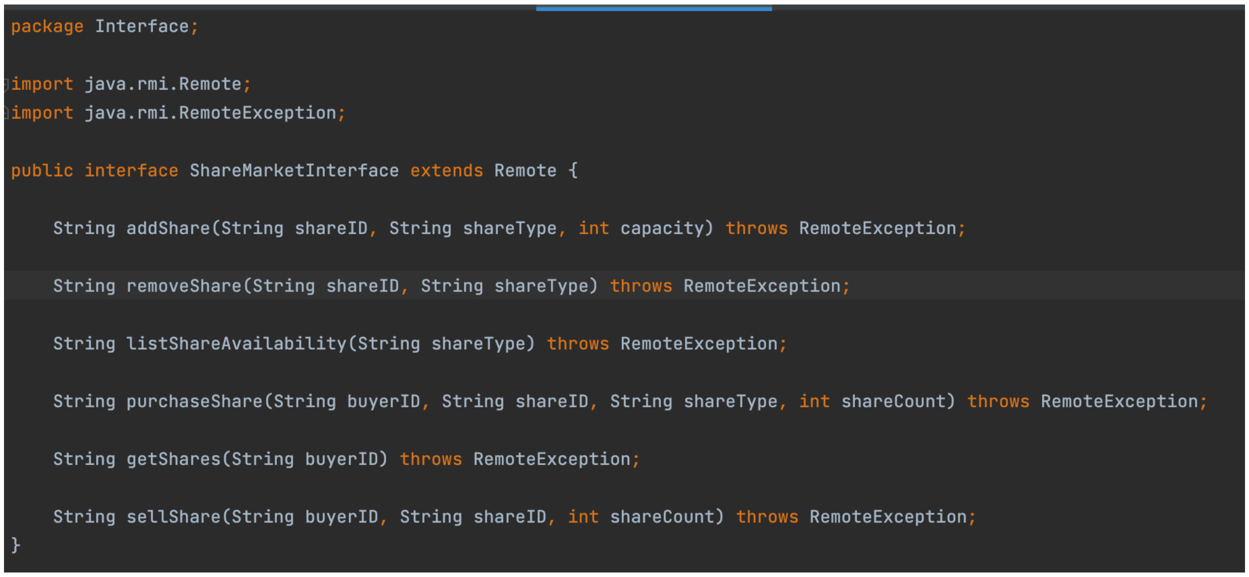
As mentioned in the assignment, we have two types of clients in this system, the first type is the buyer and the second one is the admin. Each admin could call one of the methods below:

1. Add share-> This method adds a share to the system if that share does not already exist.
2. Remove share -> This method removes a share from the system.
3. List share availability -> This method lists all the shares in the system with some information such as the remaining numbers of each share.

Buyers in this system can also perform three operations which are:

1. Purchase share -> This method lets buyers buy some shares from the system and put them in their wallets.
2. Get share -> This method shows all the shares for that buyer with the count of each share.
3. Sell share -> This method lets buyers sell several shares and return them to the system.

To implement such a system, we first need to implement the interface that defines what are the possible methods that a client can request from the servers via RMI. Then implement these functions and all the methods that are required for the UDP in the interface implementation part. Then we could create the server instance that implements a single server. After that, we create three server instances for each of our servers. Finally, we will implement the client file that will include prompts and the user interface for the clients. It worth mentioning that our system also has another file which is called logger that logs all the information necessary for both the clients and the servers. To begin with, let's start with the first file which is the interface itself. you can find this file in the src/interface/ ShareMarketInterface.java. the interface will be structured as follows:


```java
package Interface;

import java.rmi.Remote;
import java.rmi.RemoteException;

public interface ShareMarketInterface extends Remote {

    String addShare(String shareID, String shareType, int capacity) throws RemoteException;

    String removeShare(String shareID, String shareType) throws RemoteException;

    String listShareAvailability(String shareType) throws RemoteException;

    String purchaseShare(String buyerID, String shareID, String shareType, int shareCount) throws RemoteException;

    String getShares(String buyerID) throws RemoteException;

    String sellShare(String buyerID, String shareID, int shareCount) throws RemoteException;
}
```

These are the function declarations without any bodies. In the next step, we need to implement the interface and provide bodies for the methods that we declare in the interface and the methods that are required for the UDP calls from the servers. You can find the interface implementation in the ShareMarketImplimentation.java file which is in the interfaceImplementation directory. In this file, you can find the following methods and some other methods which are either a method for the UDP calls or it was required to perform some operation for one of these methods. The explanation of the main methods is as follows:

1. Admins:

The functions mentioned here are only visible to the admins and they are the only ones who can call these methods. So, for all of these methods, the first thing that I checked was whether the user was the admin of the specific server or not, if not, I printed an appropriate error message since only the admin of that server could perform these operations. Additionally, it is important to note that all of these methods should have the keyword synchronized since we are using multithreading in this system. In this file, I have also created an inner class called ShareData that stores information about the shares, including the ID, type, capacity, and count of shares for each buyer ID and some basic methods, which you can see in the code.

1.1. Add share: In this method, I check if the share that the admin wants to add already exists in the system. If it does, we print an error message since we cannot have the same share twice otherwise, we would add it to the system.

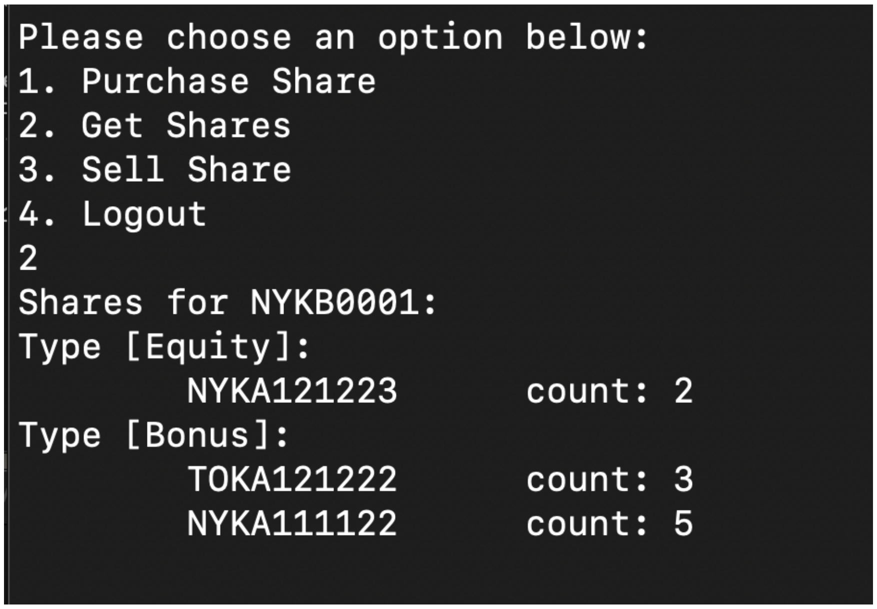
1.2. Remove share: This method is similar to the Add share method. It first checks to see if the share exists or not then if it does it attempts to remove it. One of the hard things that I have encountered here was that what if an admin wants to remove some share that exists in a buyer share list? Does it have the permission to remove that? In other words, if an admin removes a share that was purchased by a buyer, how should we handle it? I believe that since admins have more power, they could do this. So, an admin should be careful with its power since it could remove some share that belongs to the buyer.

1.3. List share availability: In this method, you just ask a server to write all of the available shares and some information about them. Then it will ask the other two servers to do the same thing using the UDP protocol. So, to use it the admin chooses a share type and the selected server will print all available shares of that share type in the entire system.

2. Buyers' methods:

   2.1. Purchase share: In this method, a buyer tries to buy a share from the servers. The first thing that we need to check is whether this buyer is from the same city that the server is in or not. If not, we should send this request using UDP to the target server to perform this operation there and update other servers. The hard part here was that it was a little challenging for me to work with UDP protocol and I was getting a lot of errors and that we have lots of conditions for buying from the servers like if you are buying from other cities, you cannot buy more than 3 shares per week. I have implemented all these conditions for both the other servers and the local server, allowing buyers to purchase shares from their city or others, as shown in the code. Another condition is to first check the availability of shares. If the requested share count exceeds the share's capacity, the buyer should only receive the maximum number of shares available. Therefore, I calculate the minimum between the share remaining and the share count before attempting to buy shares. This can have four results; one is that your purchase was successful and everything is ok so you log it. Next is that that share is full and you can't get more shares and have to wait for other buyers to sell their shares so you can buy them. The next one is that the server will check and see that you have already bought this share and you cannot buy it again due to the conditions in the assignment and you get the already registered error. And the last thing is that some problems happen and you get an error like the share that you enter to buy does not exists.

   2.2. Get share: This method shows the shares that a buyer has. So, it would check the buyer ID to see if this buyer ID does exist or not, if yes then it's just 2 nested loops that print the type of this share and then the IDs for these shares and the number of shares that the buyer has. The output for this method will look like the following image:



```
Please choose an option below:
1. Purchase Share
2. Get Shares
3. Sell Share
4. Logout
2
Shares for NYKB0001:
Type [Equity]:
        NYKA121223      count: 2
Type [Bonus]:
        TOKA121222      count: 3
        NYKA111122      count: 5
```

2.3. Sell share: This method is called by the buyers in order to sell some shares. In the first step, I checked whether the share exists on the server or not. If not, I send a request to another server using UDP to sell that share. Next step I have checked to see if the seller does have this share or not. If this share does exist then I remove it from the seller's hash map. One challenge that I encountered was to decide whether to completely delete the share or reenter it into the system for other users to buy it. I have chosen the second one since I think that makes more sense. This way I just removed it and brought it back to the system and increase the remaining so that now other buyers could buy the sold shares. Another challenge here was that we have two maps here namely allshares and buyershare which are concurrent hash maps. So, when we sell a share in this system, we should remove them from both of these maps. Since the allshares map keeps track of all shares and its structure looks like this:

```
Map<String, Map<String, ShareData>> allShares;
```

This means that for each type of share, identified by the shareID, we have corresponding information stored as ShareData, which is an inner class that holds share information. The buyershare map shows which shares each buyer has bought with the share IDs of those shares.

3. Other methods:

3.1. In this file, you can find some other methods that are methods designed for using UDP calls or simplifying basic operations. You can find more details of these methods and functions in the code.

In the next step, I created the server instance file which is in the src/server/ ServerInstance.java file of the project. This file implements a single server. First, we declare the ports (3001, 3002, 3003) for the servers and then create the remote object, registering and binding it to the registry. After that, we set the server in listening mode and each server instance will wait for incoming requests. Additionally, we have another file called server.java that creates three instances of the server instance class for each of our cities.

In the next step, we implemented the client file which is in src/client/client.java. We have created a console base terminal that prompts the user to ask for their desired operation. First, it asks for the user ID and then shows a relevant menu based on whether the user is a buyer or an admin. In this code, I checked to see if the buyer or the admin code is valid or not, if not I have printed the relative error. After the user enters its code, this code determines which city the user belongs to and connects to the corresponding server using RMI. The code then shows the operations available to the user. For example, if the user is a buyer, the options shown are buy, sell, or get shares. After the user selects its required operation, it will perform that operation by communicating to that server instance and invoking its methods.

In the last step, I created a logger that saves logs for both the servers and clients in separate files. The log files are located in the src/logs/ directory, which contains two subdirectories, one for server logs and another for client logs.

In this code, I mainly used strings for the UDP calls and separated the request string using ';' and '-'. Additionally, I mainly used concurrent hash maps, integers, and the custom classes that I have implemented and could be found in the code.

In order to make it easier to run the code I have created a Cmake file. You could simply go to the project directory and run the "make" command to compile the files. Also, you could run the "make server" command in the terminal to create the servers with the defined ports. Additionally, you can run the "make client" command to create as many clients as you need to perform the desirable tests. I have tested this code in multiple scenarios and experienced no problems. I successfully created multiple clients, made purchases from other servers, exceeded the limits, and checked for concurrency by running multiple clients simultaneously. In all of these test cases, the code has performed well.